| OF |
AD
A043487

12

RADC-TR-77-249
Final Technical Report
July 1977

RECENT DEVELOPMENTS IN THE ECL PROGRAMMING SYSTEM

Harvard University

~~Approved for public release; distribution limited~~

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

APPROVED: [illegible]

SAMUEL A. DI NITTO, Jr
Project Engineer

APPROVED: [illegible]

ALAN R. BARNUM
Assistant Chief
Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19 REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 18 RADC-TR-77-249 ✓	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER 9
4. TITLE (and Subtitle) 6 RECENT DEVELOPMENTS IN THE ECL PROGRAMMING SYSTEM		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report, 17 Sep 73 — 16 Mar 77,
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) 10 Thomas E. Cheatham Jr		8. CONTRACT OR GRANT NUMBER(s) 15 F30602-74-C-0032 new ✓
9. PERFORMING ORGANIZATION NAME AND ADDRESS Harvard University Cambridge MA 02138		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 16 55811209 17 12
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441		12. REPORT DATE 11 July 1977
		13. NUMBER OF PAGES 32 12 36p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer:
Samuel A. DiNitto (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Extensible Language, Macro-Substitution, compiling, Systems Programming Language, stepwise refinement, program transformation, code generation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Aspects of the development of the ECL Programming System are presented. ECL is built around the extensible programming language EL1. Specifically, the material deals with debugging facility, measurement aids, and optimization mechanisms in ECL, symbolic evaluation of EL1 programs, and the design and development of a subset of the full EL1 language particularly suited to systems programming.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

163 700 [illegible]

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(*When Data Entered*)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(*When Data Entered*)

EVALUATION

The contract "ECL Programming System" has resulted in this report entitled "Recent Developments in the ECL Programming System." This effort was intended to further the development of the ECL Programming System in the general areas of program optimization and software reliability.

Extensible programming languages have been with us in one form or another for almost ten years, but for reasons such as the poor efficiency of of the object programs and/or the compilers, they have been inappropriate for use in developing military systems software. RADC believes that extensible languages offer many potential benefits, most notably the ability to change constructs or to add new ones as requirements, hardware, and even programming philosophy and style change.

This effort was undertaken under TPO 5, C^3 Availability, to attempt to surmount certain of the problem areas associated with the use, or proposed use of extensible languages in military systems, and to that degree was successful. The advances in the state-of-the-art of programming environments for extensible languages made by this and similar efforts have direct application to the new proposed common DOD programming language known as DOD-1, which will have a large degree of extensibility built into it, and even to existing nonextensible languages such as the Air Force's JOVIAL.

[illegible]

SAMUEL A. DI NITTO
Project Engineer

Our central objective in developing the ECL Programming System at Harvard has been to evolve tools and techniques that reduce the cost of producing software and that increase the reliability and efficiency of the product. Achieving these goals will, in the long run, require a striking departure from conventional methods of designing and building large programs. Software engineers will need to become accustomed to the idea of deriving concrete, efficient systems from abstract, readable, demonstrably correct algorithmic specifications by a sequence of well-isolated, well-documented refinements. Development costs can be reduced through use of libraries of abstract algorithms which can be specialized to the task at hand and libraries of implementation techniques for common data and control abstractions. Maintenance costs, which often dominate the life cycle costs of software, can be reduced because the design decisions taken during a systems's construction will be spelled out in its refinement history. The dependences between such decisions will be made explicit, and the effects of changing one decision will be easy to trace. Reliability will be enhanced because the logical foundations on which the program's correctness is based, its abstract algorithms and the soundness of the refinement steps, will be exposed and manageably small. Use of proven software selected from a library also favors reliability.

Rational though such arguments may be, stepwise refinement methods will probably not be used consistently enough to have major impact until tools are available which aid in program derivation. Especially critical are optimization aids to help squeeze the unneeded generality out of an abstract program and to make it efficient while making it concrete.

This report motivates and summarizes our study of tools and techniques for stepwise refinement and optimization in conjunction with development of the ECL system. It is intended to give an overview. Detailed discussion of the individual software aids can be found in the documents cited in the text.

APPLICATION STUDIES: NEW ECL FACILITIES

Since 1974, we have studied a number of applications involving combinatorial and linear algebra and have investigated the sort of library facilities needed to support their proper development. For example, we built a set extension of ECL so that abstract algorithms involving sets can be expressed and tested at a very high level. The set package offers a selection of alternative representations for set structures from which the user can choose the most appropriate for his application.

One of our earliest experiments along this line was a matrix package. Though it has now fallen into disuse, it helped motivate several ECL improvements and new facilities which are now in regular use for software development. The matrix package provides notations for generalized selection of submatrices. For example, if M is a two-dimensional matrix, then M[i,*] and M[*,j] select the i-th row and j-th column of M, respectively. M[i THRU j,k] selects the i-th through j-th elements of column k and M[1 THRU n, 1 THRU n] is the upper left n x n submatrix of M. In each case, the selected portion is shared with the original.

The package permits users to supply the data types of matrix elements and to describe their algebraic properties. The additive and multiplicative operations to be used in matrix manipulations can be user-defined, for example, and such attributes as commutativity, associativity, and the existence of a unique inverse can be declared. Using the algebraic structure of each element type, appropriate operators are constructed for the corresponding matrix type. For instance, if the element domain forms an algebraic field, a matrix inversion operator will be provided. The extended operators are designed to exploit algebraic identities to increase their efficiency. The matrix package offers a collection of representations for sparse matrices, and matrix operators are tailored to suit the representation chosen by the user. Extended matrix operators include transitive closure, inner and outer product and matrix inversion.

Experience with the matrix package and similar applications led us to generalize ECL's mode (data type) abstraction facilities. ECL permits programmers to encapsulate data behavior in a set of user-defined functions attached to extended modes. Particular functions are invoked by the system to perform specific data dependent t[illegible]s, such as object creation and initialization, component [illegible]tion and so on. The new abstraction mechanism enables the programmer to attribute "apparent" dimensions to data structures which are not dependent on the physical sizes of the objects representing them. A sparse matrix, for example, could be implemented as a grid of linked pointer rings while its apparent dimensions are those of a rectangular array. Cascaded component selection was also made cleaner in the new facility. Suppose T, for example, is a three-dimensional structure composed of a matrix mode M whose elements are vectors of mode V. Then in the selection T[I,J,K] the selectors I and J would be processed by the user-defined selection function (USF) for matrix mode M. The resulting object and the remaining selector K would then be passed to the USF for vector mode V. The partitioning of the selectors is determined by the numbers of formal parameters of the selection functions. In this way, the USF for M need know nothing about the attributes of V -- not even its dimensionality.

The mode extension facility that resulted from these and other improvements is smoothly integrated with the base language's built-in conventions, so that the user of an extended mode need not know the details of its implementation and so that new modes can be added which interact with (e.g. convert to and from) existing types without altering their definitions. At the same time, the ECL compiler has been modified to expose extended mode attributes to compile-time manipulation (e.g. macro substitution) to insure efficient compilation of extended features. Use of the mode extension facility and the compiler are described in chapters 4 and 6 of [Manual].

The use of extended notations for clarity and modularity in the matrix and set packages made us aware of the need for other abstraction tools than those provided by ECL's data definition facility. For example, it is handy to define an iteration form like

```
FOREACH Element IN SomeArray DO
    <statement>; ...; <statement>
END
```

The intent is that the components of SomeArray be bound one by one in some order to the identifier "Element", and that the sequence of statements be performed once for each such binding. No FOREACH form exists in the EL1 base language. However, it is easy to add by syntax extension and by definition of a procedure to interpret FOREACH iterators when they are encountered in programs. An interpretive routine, nevertheless, is not always a very clear or concise description of the meaning of an expression. One would like to be able to explicate the FOREACH form by providing its meaning at a slightly lower level of abstraction. One would also like to be able to use such semantic definitions to translate high level programs successively down to the base level for interpretive debugging, analysis, or compilation.

This capability is provided by ECL's Rewrite Mechanism. A Rewrite is a pattern replacement rule. Patterns may contain match variables, distinguished by the prefix operators $$ and ??, which will be bound to arbitrary subexpressions or sublists, respectively, during matching. References to these match variables in replacements cause the corresponding bindings to be inserted in the rewritten text. For example, the meaning of the FOREACH iterator for two-dimensional matrices could be given by the rewrite

```
FOREACH $$ELEM IN $$MAT DO ??STMTS END <->
  BEGIN
    DECL M:ANY LIKE $$MAT;
    FOR I TO LENGTH(M) REPEAT
      FOR J TO LENGTH(M[I]) REPEAT
        DECL $$ELEM:ANY SHARED M[I,J];
        ??STMTS;
      END;
    END;
  END;
```

Here the "<->" operator separates the pattern on its left from the replacement, on its right. $$ELEM and $$MAT in the pattern are bound to an identifier and a matrix-valued expression, respectively; ??STMTS is bound to the list of statements comprising the body of the iterator. Although this replacement happens to be expressed entirely in the base language, it could just as well include extended notations. Rewriting rules are applied repeatedly until no matches are found, so that any number of intermediate language levels can be used. For example, matrices might be implemented as lists of lists. Then the matrix iterator would be implemented as a nested pair of iterators over lists. List iterators would in turn be implemented at a lower level, perhaps using pointer manipulations.

This raises an interesting question. Why shouldn't the iterators for matrices and lists, and other collections as well, share the same syntax, i.e. FOREACH e IN c DO s1; s2; ... END. If the rewrite mechanism were a purely syntactic macro expansion facility, there would be conflict among the rewrites defining iteration over various collection types. Rewrite patterns, however, can include predicates which augment the syntactic conditions for matching. Such a predicate may be any ECL function that maps an expression (plus other, optional inputs) to a Boolean value. All pattern predicates must return TRUE for matching to succeed. For example, if instead of $$MAT, in our matrix iterator pattern, we had written $$HAS\TYPE(MAT, MATRIX), and if HAS\TYPE is a function taking a FORM and a MODE which returns TRUE when the MODE equals the type of the FORM, then that rewrite would, without other changes, be semantically specialized to expressions of type MATRIX. A separate rewrite could be given for list iteration using the same syntax but a different type predicate.

The function HAS\TYPE, and a number of other useful expression queries, are part of a tool called the Expression Analyzer. This program performs a very weak, static interpretation of ECL program text, attempting to develop a mode and, where possible, a value or other attributes such as dimensions, for each expression in the program. Hidden semantics such as those imparted by user-defined mode behavior extensions are exposed through the creation of equivalent explicit expressions. These expressions, called <u>shadow</u> forms, are inserted along with modes and other attributes in a database which is connected to the original expressions by a hash-coded translation table. Predicates such as HAS\TYPE use the hash table developed by the Expression Analyzer and are thus quite efficient.

Shadow forms express program semantics in a standard notation which is designed to capture the meaning of any ECL expression using as few constructs as possible. Thus shadow forms are quite useful in pattern matching; rewrite patterns can drop into the shadows to make a match. The Expression Analyzer is like the front end of a compiler, and as such it can be a useful tool for direct user interrogation. In conjunction with ECL's list structure editor [Manual, chapter 5], the Analyzer allows the user to see his code from the compiler's point of view. He can browse through the text, selectively printing portions of the shadow and other attributes. He can discover passages where additional declaration would enable more efficient compilation.

The Rewrite Mechanism is documented in [Conrad1]; the Expression Analyzer, in [Holloway].

Our experience with the matrix package also suggested the need for instrumentation aids to help provide the basis for refinement decisions. The implementor needs to take measurements of program behavior when the high level version is run with sample data. In many cases, it is useful to monitor the characteristics of data items themselves at key points. To select a representation for sparse arrays, for instance, one needs to know what operations are performed most frequently (e.g. sequential versus random) and what the distribution of empty elements is likely to be. Two ECL facilities have been implemented which enable such measurements to be made during program execution by planting executable probes directly in the code. COST [Conrad2] is a tool for collecting data attributes; PROBE [Conrad3] determines execution frequencies and timing information.

PROBE carefully eliminates overhead from timings, subtracting out time spent during garbage collection as well as the time required to run the probe. It recognizes recursive calls dynamically and accounts properly for them in computing timings. It can time either compiled or interpreted routines and can break down the time spent in a routine by caller. In interpreted code it collects frequency counts which can be included as annotations in a listing of the program. These frequencies serve to highlight both sections that are executed so frequently that they deserve special care in implementation and sections that have not been exercised by the test data.

COST plants probes each of which collects the value of an ECL expression. The data collected may be numeric, Boolean, or of other types. Numeric and Boolean data are averaged and may be displayed in a variety of convenient formats when the probed program is listed. To give a very simple example, in

```
FOREACH e IN m DO
    /@ (e = 0);
         ...  ;
END
```

the COST operator "/@" records the number of times an element of matrix m is zero on entry to the loop body. After the sample run, the result would be displayed as a percentage of the total number of times the probe has been activated:

```
FOREACH e IN m DO
    \@ (e = 0) =:  62.6%;
         ...  ;
END
```

As with the Expression Analyzer, data are linked to the program text by hashing. Thus, the measurement results can be used in Rewrites that perform stepwise refinement. As a simple example, the decision whether or not to expand a particular procedure call in line might be predicated upon measured features of its arguments.

SOURCE-TO-SOURCE PROGRAM TRANSFORMATION: SYMBOLIC EVALUATION

The tools described so far have enabled us to adopt a style of constructing programs which emphasizes their abstract foundations and isolates design decisions in their realization. In building our own systems software using this style, we find it has paid off in terms of ease of initial construction and debugging, minimization of errors (particularly deep-seated design errors), and ease of maintenance when requirements change.

These tools, however, are most suitable for a user with high initiative, who understands his implementation options and the trade-offs they imply. If adaptation of general algorithms to take advantage of special circumstances is to become even semi-automatic, we will need more sophisticated analytic aids for source-to-source program transformation.

Again, the matrix application provides an example. Consider the problem of developing a special purpose matrix multiplication algorithm for tridiagonal matrices. A matrix M is said to be tridiagonal if it is square and its only non-zero elements fall on its main diagonal or on one of the two nearest subdiagonals. That is, if $|i-j| > 1$, then $M_{ij} = 0$. The specialized matrix product program is to multiply tridiagonal matrices A and B, putting the result in matrix C.

It turns out that the product C will be pentadiagonal: if $|i-j| > 2$, then $C_{ij} = 0$. Using this fact, one can devise an algorithm in which the number of scalar multiplications is reduced from $O(N^3)$, for the usual, general method, to $O(N)$, where N is the matrix dimension. The special purpose method simply clears elements of the output matrix lying off the five principal diagonals. Then it traverses those diagonals, filling in components of C. Each is the sum of at most three products of elements from A and B.

We have not yet developed optimization algorithms capable of performing specializations as sophisticated as this one automatically. However, we have begun building a Symbolic Evaluator (SE) for EL1 which will serve as the basis for furthur work on source-to-source transformation. The SE is now capable of providing the key analytical facts (such as the pentadiagonality of C) which would enable this specialization to be derived We will describe the SE and then sketch how it could be useful in the matrix product derivation.

The Symbolic Evaluator is an analyzer for EL1 programs which builds a data base of deduced facts about them. Its major components are:

(1) a symbolic interpreter, which embodies the semantics of EL1 and which builds and manages the program data base,

(2) analyzers for those control structures which give rise

to recurrence equations among program variable values, namely loops and procedures, and

(3) a simplifier/theorem prover which manipulates and reduces the expressions developed during symbolic evaluation.

The SE's symbolic interpreter begins where the Expression Analyzer leaves off. To every program variable it attaches a location description, which contains the mode of the variable and some representation for values of the variable in every program context which lies within its scope. Variable values are manifest constants, where possible, and otherwise they are symbolic expressions (SEXPRs) composed of pure arithmetic, logical, and structural functions. Sharing patterns among locations, including conditional patterns, are carefully tracked by the symbolic interpreter and are used to account for all possible side effects when the value of a particular variable is being fetched. A context graph is constructed which reflects each branch of control in the program and which records the symbolic predicate that conditions each branch. The conjoined predicates leading from the entry of the program to a particular context comprise the "path condition" of that point. Path conditions are extremely useful in reasoning about variable behavior (yet most compilers pay no attention to them).

When it encounters a loop, the symbolic interpreter processes the loop body, obtaining for each variable an expression representing the effects of a general cycle of the loop as a function of the values of variables at the beginning of the cycle. Then the loop analyzer is called in to try to solve the resulting recurrence equations, producing, when possible, closed-form expressions for loop variable values as a function of the cycle number. The loop analyzer also determines the symbolic condition for loop exit and attempts to derive the total number of cycles of the loop. This result is then used to produce values for variables after loop exit.

A component of the loop analyzer called the "row solver" specializes in solving the recurrence equations which arise for array variables in loops. Here the problem is complicated by the uncertainty as to which component location is being affected by a given array assignment. Using only a general recurrence equation solving method, the row solver is frequently able to describe each output array element as a function of initial variable values only. As a simple illustration, the matrix transpose loop:

```
FOR I TO N REPEAT
  FOR J TO I-1 REPEAT
    SWAP(M[I,J], M[J,I])
  END
END
```

results in the following expression as the value for M:

array(<i,j>, <N_1,N_1>, M_1[j, i])

Here, "array" is a structure-valued SEXPR function which yields an array, in this case two-dimensional, with both dimensions equal to N_1, the symbolic value of N. i and j are dummy parameters ranging between 1 and their corresponding dimensions. The third argument describes an array element at position <i,j>. M_1 is the symbolic value of M at the beginning of the outer loop. In other words, this expression describes an N x N array which is the transpose of the original M.

The procedure analyzer (which has been designed but not yet implemented) handles non-recursive and certain simple recursive procedures. It analyzes each once in isolation and then applies the resulting analysis to each point of call. The treatment of recursive procedures again produces recurrence relations to be solved, and the procedure analyzer shares the loop analyzer's tools for doing so.

The simplifier/theorem prover attempts to reduce SEXPRs to manifest constants. In the process, it places them in a normal form so that those with equal values will tend to have the same representation. Syntactically equivalent SEXPRs are replaced, using a hash coded translation table, by a unique list structure representative. This greatly speeds up SEXPR equality checking and allows the simplifier to attach attributes to SEXPRs by direct hashing. In addition to the conventional rules for arithmetic and logical reduction, the simplifier includes a "linear solver". The linear solver takes conjunctions of linear inequalities involving rational variables and tests them for consistency. If they are inconsistent, it replaces the conjunction by the constant "false". Otherwise, it tries to derive strict equalities, or bounds on linear combinations of variables implied by the original conjucntion. Finally, the logical simplification algorithms include a resolution theorem prover for the ground case. We expect to extend the theorem prover to assertions involving quantifiers in the near future.

A more detailed description of the Symbolic Evaluator will be found in [Cheatham1, Cheatham2, Townley].

To return to the problem of deriving a specialized product algorithm for tridiagonal matrices, let us consider how we would use the SE to motivate specializations. From the general matrix multiplication loop, the SE derives the value for the output matrix C:

```
 C = array(<i,j>, <N,N>, finite\sum(k, 1, N, A[i,k]*B[k,j]))
```

(Here the expression finite\sum(k, L, U, f(k)) means

```
    f(L) + f(L+1) + ...  + f(U)
```

if $U \geq L$ and is zero otherwise.) The tridiagonal condition is expressed as the SEXPR

```
   M[i,j] = cond(-1 le i-j and i-j le 1, M[i,j], 0)
```

(Here cond(p, x, y) may be read 'if p then x else y'.) Using this fact for the input operands A and B, the simplifier would produce

```
   C = array(<i,j>, <N,N>,
             finite\sum(k, 1, N,
                        cond(-1 le i-k and i-k le 1 and
                               -1 le k-j and k-j le 1,
                             A[i,k] * B[k,j],
                             0)))
```

When the linear solver is applied to the predicate of the conditional, it produces the derived conjunction -2 le i-j and i-j le 2. Only when this condition is satisfied will there be non-zero contributions to C[i,j]. In other words, C will be pentadiagonal.

Since only five values of i-j are involved, we can use case analysis to try to eliminate the finite\sum and thereby eliminate a loop in the program. With i-j chosen to be -2, the linear solver determines that the conditions for non-zero terms are k = i+1 and i le N-2. Thus C[i,j] = A[i,i+1] * B[i+1,i+2] in this case. The case when i-j = 2 is similar. When i-j is fixed at -1, the relations i le N-1 and i le k and k le i+1 result. So in this case the sum expands to

```
   C[i,j] = A[i,i] * B[i,i+1] + A[i,i+1] * B[i+1,i+1]
```

The case i-j = 1 is similar. Finally, when i = j, we have

```
   max(1, i-1) le k and k le min(N, i+1)
```

Since the spread of possible values for k (and hence the number of non-zero terms) is at most three, the program transformer might opt for complete expansion of the finite\sum.

Thus, we derive the new symbolic expression for C:

```
C =
  array(<i,j>, <N,N>,
        CHOOSE
          i-j lt -2 => 0;
          i-j = -2 => A[i,i+1] * B[i+1,i+2];
          i-j = -1 =>
            A[i,i] * B[i,i+1] + A[i,i+1] * B[i+1,i+1];
          i-j = 0 and i = 1 and N = 1 => A[1,1] * B[1,1];
          i = j and i = 1 and N gt 1 =>
            A[1,1] * B[1,1] + A[1,2] * B[2,1];
          i = j and i gt 1 and i = N =>
            A[N,N-1] * B[N-1,N] + A[N,N] * B[N,N];
          i = j and i gt 1 and i lt N =>
            A[i,i-1] * B[i-1,i] + A[i,i] * B[i,i] +
              A[i,i+1] * B[i+1,i];
          i-j = 1 =>
            A[i,i] * B[i,i-1] + A[i,i-1] * B[i-1,i-1];
          i-j = 2 => A[i,i-1] * B[i-1,i-2];
          i-j gt 2 => 0;
        END);
```

(Here the expression CHOOSE p_1 => e_1; p_2 => e_2; ... END is a multi-arm conditional SEXPR in which the predicates p_1, p_2, ... are mutually exclusive.) From this symbolic value, the transformer can plausibly synthesize a new loop to produce the appropriate value of C. In fact, it can use several loops: doubly nested loops for the cases in which i and j have no fixed difference (e.g., i-j lt -2), and single loops when j is a function of i. No loop at all need be used when i and j are constants. Since, by construction, the cases are mutually exclusive, the order of the loops is immaterial.

```
   FOR i TO N REPEAT
     FOR j TO N REPEAT
       i-j LT -2 -> C[i,j] <- 0
     END
   END;
   FOR i TO N-2 REPEAT
     C[i,i+2] <- A[i,i+1] * B[i+1,i+2];
   END;
   FOR i TO N-1 REPEAT
     C[i,i+1] <-
```

```
      A[i,i] * B[i,i+1] + A[i,i+1] * B[i+1,i+1];
  END;
  N = 1 -> C[1,1] <- A[1,1] * B[1,1];
  N GT 1 -> C[1,1] <- A[1,1] * B[1,1] + A[1,2] * B[2,1];
  N GT 1 ->
    C[N,N] <- A[N,N-1] * B[N-1,N] + A[N,N] * B[N,N];
  FOR i FROM 2 TO N-1 REPEAT
    C[i,i] <-
      A[i,i-1] * B[i-1,i] + A[i,i] * B[i,i] +
        A[i,i+1] * B[i+1,i];
  END;
  FOR i FROM 2 TO N REPEAT
    C[i,i-1] <-
      A[i,i] * B[i,i-1] + A[i,i-1] * B[i-1,i-1];
  END;
  FOR i FROM 3 TO N REPEAT
    C[i,i-2] <- A[i,i-1] * B[i-1,i-2];
  END;
  FOR i TO N REPEAT
    FOR j TO N REPEAT
      i-j GT 2 -> C[i,j] <- 0
    END
  END;
```

This is a much more efficient program than the unspecialized version. Of course, further improvements can be made to tidy it up. Loop reduction could be used to simplify the first loop to

```
FOR i FROM 4 TO N REPEAT
  FOR j TO i-3 REPEAT
    C[i, j] <- 0
  END
END
```

for example. Loop fusion could also be used to combine those loops with compatible parameter ranges.

The point of presenting this example has not been to represent the SE as a complete tool for program transformation. It is not, and we appreciate the difficulty of automating such transformations in general. We are confident, however, that the Symbolic Evaluator provides a suitable basis on which to develop source-to-source specialization techniques, whether user-guided or completely mechanical.

OPTIMIZED MACHINE CODE GENERATION

However sophisticated our source-to-source optimization techniques become, they will still need to be complemented by compilation algorithms which produce efficient machine code. We have therefore implemented an optimization pass for ECL's compatible compiler. Called IMPROVE, it walks the program tree output by ANALYZE (the analysis phase) and produces a modified tree for input to CODE (the code generation phase). We have chosen to concentrate on the problem of eliminating redundant subexpressions, and on the so-called "invalidation problem" for potentially available expressions. Other machine code optimization issues, including code motion, dead code elimination, and optimization of register assignment, have been examined in building the SPECL compiler for a systems programming subset of ECL. SPECL, which will be described below, also uses ANALYZE and IMPROVE as a front end.

The IMPROVE Phase

The invalidation problem arises because sharing patterns among variables and data structures permit hidden side effects to destroy the availability of expressions not manifestly affected. For example, suppose X and Y are LIST variables, and consider the sequence

```
... HEAD(TAIL(X)) ...
  Y <- NEWLIST;
... HEAD(TAIL(X)) ...
```

Unless it can be shown that Y is shared neither with X nor with TAIL(X), the value of HEAD(TAIL(X)) must be recomputed; that is, it is invalidated by the possible side effect. In a language with the rich potential for sharing that ECL has, the relationship between X and Y may be quite obscure, particularly if they are formal parameters or free variables of the procedure being compiled. This is sometimes called the "aliasing problem."

Our approach in IMPROVE is to cope as effectively as we can with the invalidation problem without engaging in intense analysis. Fortunately, a significant number of potentially damaging side effects can be ruled out on very simple grounds of scope, storage status (heap versus stack) or data type.

Two types of common subexpressions are distinguished: pure values and proper objects. Pure values are expressions like X + Y and LENGTH(A) whose values are transient; that is, they occupy no identifiable place in storage. The availability of a pure value for reuse ends if any of is subexpressions is redefined. When this type of redundant expression is removed, its replacement acts like a "runtime constant": storage for the value is allocated at procedure entry and is initialized when the "parent" or defining occurrence is evaluated. Thereafter it may be referenced but not changed.

Proper object expressions are those with identifiable, reusable storage locations, such as A.FIELD[I] and VAL(P). When found redundant, they too are replaced by references to temporaries. But proper object temporaries behave like shared variables, rather than constants. For example, the sequence

```
        DECL A:RECORD;
             ...
        PRINT(A.FIELD[I]);
             ...
        A.FIELD <- NEWFIELD;
        PRINT(A.FIELD[I]);
```

would be replaced, in effect, by

```
        DECL A:RECORD;
             ...
        DECL T1:ANY SHARED A.FIELD;
        DECL T2:ANY SHARED T1[I];
        PRINT(T2);
             ...
        T1 <- NEWFIELD;
        PRINT(T2);
```

Care must be taken to see that stack space for proper object temporaries is allocated at the same block level as the parent object (RECORD A in the example above). If it were allocated at the procedure level, for example, dangling references could outlive an object allocated on the stack, leading to confusion for the storage manager.

As IMPROVE walks the program tree, it maintains an Available Expression List (AEL). Only simple expressions using built-in operators are included; control expressions are not. Each expression listed in the AEL is classified by mode, by context of creation, by expression type (pure or proper), and if proper, by scope and locale (local versus global, stack, heap or unknown). As a new simple expression is walked, it is tested for redundancy with existing available expressions. Two expressions match when

(1) they have the same operator, and
(2) they have the same number of operands, and
(3) each pair of corresponding operands match recursively.

If the new expression matches none of the AEL members it is made available by adding it to the list. Otherwise it is linked to the existing expression subtree.

Various transitions and events in IMPROVE's walk of the tree cause it to scan the AEL and prune invalid elements. A change of context, for example, may cause some expressions to become "unavailable" while the validity of others is re-established. A control excursion, such as a call to a procedure whose effects are unknown, forces all available expressions to be dropped except those depending strictly on "hidden" local names. Pruning expressions at an assignment is essentially a pattern matching process, using the object descriptions stored with AEL members. When the object being assigned is well defined, e.g. an unshared local variable of known mode, affected expressions can be pruned quite precisely. In less well resolved cases of course, more AEL elements may have to be removed to guarantee correctness. For example, VAL(INTPTR) <- I + 1 invalidates not only the expression being redefined but also all INT expressions not known to reside on the stack.

Even so,the inexpensive optimization schemes used in IMPROVE provide an efficient complement to the more powerful tools that will become available through symbolic evaluation.

The SPECL Compiler

The SPECL (Systems Programming in ECL) project is intended to extend the use of ECL into areas normally reserved for so-called "implementation languages." SPECL is a dialect of ECL that can be compiled to stand-alone machine code that runs without the support of ECL's runtime facilities. It offers the opportunity to "contract" ECL for special applications, since SPECL-produced code can be augmented by just the runtime support (such as storage management or I/O) that is needed.

SPECL also offers access to the implementation of operators and the choice of underlying data representations at the hardware level. For example, suppose a programmer wants to implement doubly linked lists using a minimum of storage for the links. One trick is to give each element a single link field containing the bitwise exclusive-or of the address of its predecessor with that of its successor. Given pointers to any two successive elements it is then a simple matter to move forward or backward along the list.

SPECL is ideal for such an application since it permits the user to manage storage as he chooses and allows him to give machine code definitions for the necessary pointer operations. Access to the hardware level is isolated in code generation templates called Compiler Control Expressions (CCE). A CCE consists of a specification of a set of methods that may be used by the compiler to translate the application of an operator to a set of operands. Included in the CCE are constraints on the order of evaluation of operands; information regarding which, if any, of the operands and the result are proper objects; whether operands are modified or merely referenced; and a set of methods of generating code to evaluate the operator. The methods include information about the form and dispositions of operands and result, side effects, additional registers needed, the cost of code generated, and of course, the code template itself.

SPECL compilation begins with the same two phases, ANALYZE and IMPROVE, used by the ECL interpreter-compatible compiler. The two compilers thus share the same redundancy recognition scheme. The rest of the SPECL compiler consists of five passes. They are, first, an initial pass (LAB1) in which the program tree is labelled with information needed for later passes, an order of evaluation is chosen, and an initial selection of methods for operator compilation is made. The second pass, LAB2, is a reverse-order pass, made to determine the lifetime of values of variables and operator results, and the distance between references to those values. This information is used by the two allocation passes, ALLOC1 and ALLOC2 which determine which items are allocated to registers, interpose the necessary loads, stores and other adjustments, and generate the conflict network used by the register assignment program ASSIGN. After assignment a final pass, EMIT, generates the actual instructions.

Starting from a trial ordering of the expressions in each context (straight-line program section), the temporary-minimization procedure examines the variation in storage requirements over each context. The vicinities of peaks in the requirements are scanned for target positions that meet a simple numeric criterion based on the temporary usage and result size of the neighboring computations. Expressions are moved to these favorable positions in order to reduce overall storage use. Subtrees free of common subexpressions move as units, subject to safety constraints. Most of the time used by this process is actually spent in recalculating the temporary requirements after a move.

The register assignment algorithm first makes a live-dead analysis of variables and common sub-expresseions and determines the minimum distance to next use for these items. This gives the information needed to perform register allocation optimally in straight-line code. Assignment is straightforward in such branch-free regions as well. With control structure, however, the problem is to match assignments at branches and join points. Heuristics are used to reduce the computation from a "try all assignments" approach.

In summary, then, SPECL extends to the hardware level the methodology that characterizes program design using ECL. Users are permitted to become involved in the optimization of their programs, and they need not forsake good structure to achieve highly efficient performance.

Further details on the SPECL compiler can be found in [Udin]. At present the basic translator/optimizer is essentially complete as described. Missing are portions for handling certain specific EL1 constructs such as THUNKS and MARK-RETURN. Most of the actual operators of EL1, other than control structures and assignment, have been left to be implemented as Compiler Control Expressions.

SYNCHRONIZATION OF SYSTEMS OF CONCURRENT PROCESSES

Coordination of multiple concurrent processes is an area of software engineering in which automated aids to program reliability are especially critical. Correct synchronization of complex systems of interacting parallel processes is a tricky problem under the best of circumstances. Conventional multiprogramming facilities often aggravate the problem by forcing programmers to think in terms of low level synchronization primitives within the individual processes. A more natural approach is to view synchronization in terms of the state of the system as a whole. One would like to construct a model of the system reflecting all of its significant states and the conditions for state changes, and then generate code from that model to be used by individual sequential processes in effecting state transitions.

We have constructed an experimental facility which implements this centralized approach to synchronization with the ECL system. SYNVER is a tool which allows designers of multi-process systems to specify the coordination of processes in a high level specification language (an extension of EL1). SYNVER verifies the consistency of the user's model, checking, for example, for deadlock situations. It then generates synchronization code to be used by the individual processes. These synchronization functions are expressed in terms of ECL's multi-path control primitives [Prenner].

SYNVER has been successfully applied to a wide variety of synchronization paradigms. A complete description of SYNVER with examples of its use is contained in [Griffiths].

Because code generated by SYNVER depends on the control interpreter of the ECL system, a relatively high level facility, it is interesting to consider how SYNVER-created synchronization functions could be translated to use very low level (and easily implemented) synchonization primitives, such as Dijkstra's semaphores. Unfortunately, brute force translation of such functions to the semaphore level gives very inefficient results. However, the existence of a high level specification including invariant assertions made by the user, plus the knowledge of constraints on the functions generated by SYNVER, permits significant optimizations to be performed on semaphore code produced from them. We have studied algorithms for generation of optimized semaphore code, and have described them in detail in [Steele].

ARPANET ACCESS FROM ECL

To widen ECL'S applicability to probelms of interest to the ARPANET community, we have extended ECL's I/O facilities to permit general use of network connections without leaving the system. Thus interfaces with remote databases or program tools can be constructed easily within ECL. To test the new facility and provide an example of its use, the TELNET protocol has been implemented in ECL.

REFERENCES

[Manual] ECL Programmer's Manual. Technical Report 23-74, Ctr. for Res. in Comp. Tech., Harvard Univ.

[Conrad1] Conrad, W.R. RERITE User's Guide. Technical Report, Ctr. for Res. in Comp. Tech., Harvard Univ.

[Conrad2] Conrad, W.R. COST User's Guide. Technical Report, Ctr. for Res. in Comp. Tech., Harvard Univ.

[Conrad3] Conrad, W.R. PROBE User's Guide. Technical Report, Ctr. for Res. in Comp. Tech., Harvard Univ.

[Holloway] Holloway, G.H. User's Guide to the Expression Analyzer and Query Facility. Technical Report, Ctr. for Res. in Comp. Tech., Harvard Univ.

[Cheatham1] Cheatham, T.E., Jr. and Judy A. Townley. Symbolic Evaluation of Programs - A Look At Loop Analysis. Proc. of SIGSAM Symposium, Aug. 1976.

[Cheatham2] Cheatham, T.E. Jr., G.H. Holloway and Judy A. Townley. A General Approach to Symbolic Evaluation. Ctr. for Res. in Comp. Tech., Harvard Univ, to appear.

[Cheatham3] Cheatham, T.E., Jr. and Judy A. Townley. A Look At Programming and Programming Systems. Advances in Computers, Vol. 14, 1976.

[Davis] Davis, M.W. Some Methods Involving Sharing for Static Program Analysis. Technical Report. Ctr. for Res. in Comp. Tech., to appear.

[Townley] Townley, Judy A. A Symbolic Interpreter for EL1. Technical Report, Ctr. for Res. in Comp. Tech., Harvard Univ.

[Udin] Udin, David. Systems Programming Languages. Technical Report, Ctr. for Res. in Comp. Tech., Harvard Univ.

[Prenner] Prenner, C.J. Multi-path Control Structures for Extensible Languages. Technical Report. Ctr. for Res. in Comp. Tech., Harvard Univ.

[Griffiths] Griffiths, P.P. SYNVER: An Automatic System for the Synthesis and Verification of Synchronous Processes. Technical Report 20-75, Ctr. for Res. in Comp. Tech., Harvard Univ.

[Steele] Steele, G.L., Jr. Generation of Optimized Semaphore Synchronization Code. Technical Report 21-75, Ctr. for Res. in Comp. Tech., Harvard Univ.